Scheduled to appear in The Journal of Supercomputing, , 1–25 ()

MCGS: A Modified Conjugate Gradient Squared Algorithm for Nonsymmetric Linear Systems

MUTHUCUMARU MAHESWARAN maheswar@cs.umanitoba.ca
Department of Computer Science, University of Manitoba, Winnipeg, MB R3T 2N2, Canada

KEVIN J. WEBB, AND HOWARD JAY SIEGEL {webb,hj}@ecn.purdue.edu
Parallel Processing Laboratory, School of Electrical and Computer Engineering, 1285 Electrical Engineering Building, Purdue University, West Lafayette, IN 47907-1285, USA

Editor: Hamid R. Arabnia

Abstract. The conjugate gradient squared (CGS) algorithm is a Krylov subspace algorithm that can be used to obtain fast solutions for linear systems ($\mathbf{Ax} = \mathbf{b}$) with complex nonsymmetric, very large, and very sparse coefficient matrices ($\mathbf{A}$). By considering electromagnetic scattering problems as examples, a study of the performance and scalability of this algorithm on two MIMD machines is presented. A modified CGS (MCGS) algorithm, where the synchronization overhead is effectively reduced by a factor of two, is proposed in this paper. This is achieved by changing the computation sequence in the CGS algorithm. Both experimental and theoretical analyses are performed to investigate the impact of this modification on the overall execution time. From the theoretical and experimental analysis it is found that CGS is faster than MCGS for smaller number of processors and MCGS outperforms CGS as the number of processors increases. Based on this observation, a set of algorithms approach is proposed, where either CGS or MCGS is selected depending on the values of the dimension of the $\mathbf{A}$ matrix (N) and number of processors (P). The set approach provides an algorithm that is more scalable than either the CGS or MCGS algorithms. The experiments performed on a 128-processor mesh Intel Paragon and on a 16-processor IBM SP2 with multistage network indicate that MCGS is approximately 20% faster than CGS.

Keywords: algorithm scalability, conjugate gradient squared, modified conjugate gradient squared, Intel Paragon, IBM SP-2, MIMD, synchronization.

1. Introduction

This is an application-driven study of solutions to linear systems of equations ($\mathbf{Ax} = \mathbf{b}$) on MIMD parallel machines. The application being considered is the *finite element method* (*FEM*) modeling of open-region electromagnetic problems in the frequency domain [7, 8]. The matrices obtained in this problem are very large, very sparse, nonsymmetric, and have complex-valued elements.

For the 2-D physical examples considered, first-order (linear) node-based functions over triangular elements are used. The resulting $\mathbf{A}$ matrix is unstructured, with the non-zero entries dictated by the global node numbering. The correspondence between the node numbering and the sparsity pattern of $\mathbf{A}$ is illustrated in Figure 1 for a simple 2-D mesh. Figure 1(a) shows the connectivity among the nodes of the mesh and Figure 1(b) shows the sparsity pattern of the resulting $\mathbf{A}$ matrix. Note that while the sparsity pattern of $\mathbf{A}$ is symmetric, the actual matrix element values are not. In contrast, quadrilateral elements, which are commonly

used in finite difference representations, result in a structured, multi-diagonal **A** matrix. Triangular elements provide greater flexibility in the representation of the geometry, but create the need for solution procedures that do not rely on an multi-diagonal **A** matrix. A similar **A** matrix will result for any wave equation problem that is described by a differential equation. The challenge is to be able to solve very large order problems effectively.

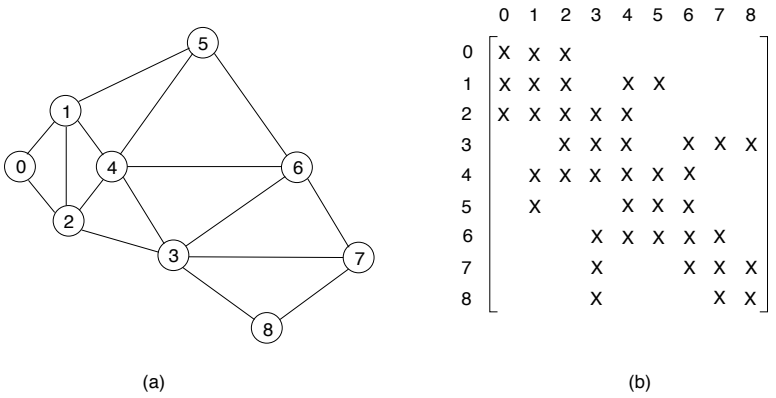

Figure 1. A simple 2-D mesh and the corresponding portion of the **A** matrix.

The *conjugate gradient squared* (*CGS*) algorithm [14] is used for the solution of the linear system. This algorithm can provide fast solutions, even though the convergence pattern is often non-uniform. This study focuses on the performance and scalability of this algorithm on MIMD machines. Synchronization and communication are two factors that introduce significant overhead when this algorithm is implemented on a parallel machine. The communication overhead is dependent on the structure of **A**, i.e., the sparsity pattern of **A**. Therefore, matrix reordering techniques can be used to reduce this overhead. The synchronization overhead depends on the number of vector-vector inner products performed per iteration of the algorithm. For MIMD machines, the synchronization cost rises significantly with increasing machine size. Hence, for scalable MIMD implementations, the amount of synchronization has to be minimized.

This paper proposes a *modified CGS* (*MCGS*) algorithm where the synchronization overhead is effectively reduced by a factor of two. This is achieved by changing the computation sequence in the CGS algorithm. An approximate theoretical complexity analyses and experimental studies have been done to investigate the impact of the modification on the overall execution time of the CGS algorithm. The approximate complexity analyses using a mesh-connected model indicates that for larger machine sizes the performance of the MCGS algorithm may be up to 34% better than that of the CGS algorithm, depending on the machine architecture. For smaller matrix sizes, CGS performs better than MCGS. The experimental studies on a 128-processor Intel Paragon reveals that MCGS is at least 20% better than

the CGS for larger number of processors. The experiments are also performed on a 16-processor IBM SP2.

Because neither algorithm is better than the other for all values of input data sizes and system parameters, a set of algorithms approach is presented (e.g., [13, 18]). This provides a scalable solution scheme for $\mathbf{Ax} = \mathbf{b}$. Conditions for choosing a particular algorithm depending on input data and system parameters are also provided. Conditions such as the one developed here to choose between CGS or MCGS depending on the system parameters are also useful in the area of heterogeneous computing (*HC*) mapping systems [9]. In HC mapping, the input data (e.g., matrix size) remains fixed, but the system parameters are varied, i.e., the mapping system estimates the performance on different machines and executes the application on the machine that is expected to yield the best performance. To obtain the best mapping, it is necessary for the HC mapping systems to have information such as those provided by the conditions to select either MCGS or CGS depending on system parameters.

The solution of $\mathbf{Ax} = \mathbf{b}$ is a time consuming step in the FEM modeling of many problems from diverse areas such as fluid dynamics, structures, and atmospherics. Therefore, a lot of work has been done in designing iterative algorithms and their parallel implementations for solving $\mathbf{Ax} = \mathbf{b}$. Some of the widely used iterative algorithms are the Krylov subspace algorithms. In general, these algorithms provide fast and robust solutions for $\mathbf{Ax} = \mathbf{b}$.

Dazevedo et al. [3] developed two reformulations for the *conjugate gradient* (*CG*) algorithm that reduce the synchronization overhead associated with the parallel implementations of the generic CG algorithm. Meurant [10] and Saad [11] also discuss the reduction of the synchronization overhead in the parallel implementations of the CG algorithm. The CG algorithm is applicable for symmetric positive definite (SPD) $\mathbf{A}$ matrices. The work presented in the following sections extends these ideas to reduce the synchronization overhead in the CGS algorithm, which is applicable to nonsymmetric $\mathbf{A}$ matrices.

In Section 2, the open-region electromagnetic problem is briefly examined. General strategies used for solving linear systems with complex elements are discussed in Section 3. The machine model and the notation used to parameterize the algorithms are presented in Section 4. Section 5 examines the issues involved in parallelizing some of the basic linear algebra subroutines needed for implementing a Krylov algorithm. A modified CGS algorithm is provided in Section 6 and this algorithm is compared with the CGS algorithm in Section 7. In Section 8, the experimental results are presented.

2. Problem Definition

This section is a brief overview of the physical problem. See [7, 8] for details.

Progress in micro-machining of optical diffractive elements has sparked an interest in the study of wavelength and subwavelength structures in the last few years. A rigorous modeling of these devices demands the solution of the underlying elec-

tromagnetic equations. The FEM is used here to discretize the wave equation. The resulting set of linear equations then needs to be solved.

Consider a scalar wave equation for a 2-D problem with a time dependency $e^{j\omega t}$ for the fields. Let u be the total, unknown, complex-valued scalar field, g be the magnetic or electric source inside the computational domain, and k_0 be the wave number in free space. Also, let p be the material permittivity, q be the material permeability, ε_0 be the permittivity of the free space, and μ_0 be the permeability of the free space. The scalar wave equation in the frequency domain is

$$\bigtriangledown \cdot \ (p \ \bigtriangledown u) + k_0^2 \ q \ u = g, \tag{1}$$

with $k_0^2 = \omega^2 \mu_0 \ \varepsilon_0, \ u = H_z, \ p = \varepsilon_r^{-1}$ and $q = \mu_r$
for transverse electric (TE) polarization or $u = E_z, \ p = \mu_r^{-1}$
and $q = \varepsilon_r$ for transverse magnetic (TM) polarization.

A general open-region electromagnetic scattering problem with an artificial boundary $\partial\Omega$ for TM polarization is shown in Figure 2, where the superscript *inc* denotes the incident field components, the superscript *tot* denotes the total field components, the superscript s denotes the scattering field components, and *pec* denotes a *p*erfect *e*lectric *c*onductor. Let B_i represent the basis functions and D_1, D_2, and D_3 be coefficients that originate from radiation boundary conditions or absorbing boundary conditions at the outer boundary [8]. Furthermore, let u^{inc} denote the incident field. Then, the discretization of the corresponding variational formulation results in a system of linear equations:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2}$$

with A_{ij} and b_i represented by

$$\begin{aligned} A_{ij} \ &= \ \int_\Omega \int \left\{ p \nabla B_i \cdot \nabla B_j - k_0^2 \ q \ B_i \ B_j + g \ B_i \right\} ds \\ &\quad - \oint_{\partial\Omega} \left\{ D_1 \ B_i \ B_j + \left(B_i \ D_2 - \frac{\partial}{\partial m} \left(B_i \ D_3 \right) \right) \frac{\partial B_j}{\partial m} \right\} \ dm \end{aligned} \tag{3}$$

$$b_i \ = \ \oint_{\partial\Omega} B_i \left(\frac{\partial u^{inc}}{\partial n} - D_1 \ u^{inc} - D_2 \ \frac{\partial u^{inc}}{\partial m} - D_3 \ \frac{\partial^2 u^{inc}}{\partial m^2} \right) \ dm \tag{4}$$

The $\mathbf{A}$ matrix is of size N [7], corresponding to the number of free variables (number of unknowns) within the domain. Because of the local linear expansion and testing functions, the $\mathbf{A}$ matrix typically has a fill factor less than one percent.

3. Linear Equation Solution

Various approaches are used in the literature for solving linear systems with complex coefficient matrices [5]. One approach is to premultiply $\mathbf{A}\mathbf{x} = \mathbf{b}$ by the hermitian matrix $\mathbf{A}^H$ ($\mathbf{A}^H = (\mathbf{A}^*)^T$) [15], which is also known as the normal equations

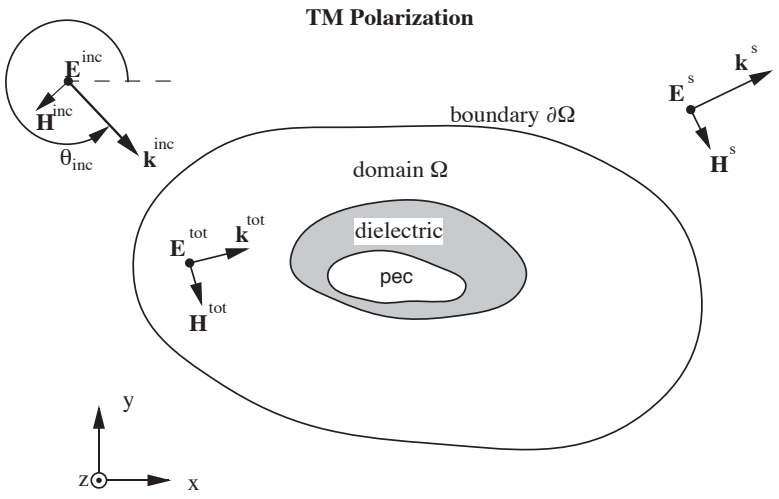

Figure 2. General scattering problem in unbounded region: artificial boundary $\partial\Omega$ and different field components for TM polarization.

method. This results in a linear system $\tilde{\mathbf{A}}\mathbf{x} = \tilde{\mathbf{b}}$, where $\tilde{\mathbf{A}} = \mathbf{A}^H\mathbf{A}$ and $\tilde{\mathbf{b}} = \mathbf{A}^H\mathbf{b}$. $\tilde{\mathbf{A}}$ has real-valued elements and $\tilde{\mathbf{b}}$ can have complex-valued elements. Because $\tilde{\mathbf{A}}$ is real, the system can be split into two systems of equations:

$$\tilde{\mathbf{A}} \begin{bmatrix} Re(\mathbf{x}) \\ Im(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} Re(\mathbf{b}) \\ Im(\mathbf{b}) \end{bmatrix} \tag{5}$$

For some $\mathbf{A}$ matrices this method can be very efficient, e.g., when $\mathbf{A}$ is close to unitary ($\mathbf{A}^H\mathbf{A} \approx \mathbf{I}$). However, the normal equations method often leads to systems with poor convergence properties. This is due to the condition number of $\tilde{\mathbf{A}}$ being approximately the square of the condition number of $\mathbf{A}$. For many complex coefficient systems that occur in practical applications, the normal equations method has been observed to be inefficient [5] and so is not considered in this study because of the structure of the $\mathbf{A}$ matrix involved.

Another approach is to split the complex system into real and imaginary components:

$$\mathbf{A}^* \begin{bmatrix} Re(\mathbf{x}) \\ Im(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} Re(\mathbf{b}) \\ Im(\mathbf{b}) \end{bmatrix}, \text{ where } \mathbf{A}^* = \begin{bmatrix} Re(\mathbf{A}) & -Im(\mathbf{A}) \\ Im(\mathbf{A}) & Re(\mathbf{A}) \end{bmatrix} \tag{6}$$

This process doubles the size of the coefficient matrix on each dimension, i.e., if $\mathbf{A}$ is $N \times N$ then the resulting coefficient matrix, $\mathbf{A}^*$, is $2N \times 2N$. This method is not very efficient, because the spectral properties of the resulting coefficient matrix are less suitable for Krylov iterative methods than that of the original matrix [4].

Hence, in this study, real Krylov iterative algorithms are adapted to solve complex linear systems. Although the overall algorithm remains the same, there are changes to the basic operations such as the inner products.

4. The Machine Model and Terminology

The analysis of algorithms presented in this paper is based on the distributed memory MIMD machine model. The machine has P processors and each processor is paired with a memory module to form a *processing element* (*PE*). The PEs are connected by an interconnection network. In the experiments reported here, all PEs execute the same code, i.e., the *single program multiple data* (*SPMD*) model is used. The PE's memory is used to store both instructions and data. If a PE needs data stored in a remote PE then it is retrieved by message passing.

The algorithm complexity analysis performed in the next section is for a mesh-based distributed memory MIMD machine model. To simplify the analysis and parameter measurement, the communication operations on the Intel Paragon are modeled without considering the wormhole routing [2] that is used in the Paragon. The computation is modeled by counting the floating point operations (FLOPs) at the source code level (the C language is used). The following notation is used in this paper: (1) P: number of PEs, (2) N: dimension of the $\mathbf{A}$ matrix, (3) k: number of non-zero elements per row of the $\mathbf{A}$ matrix (for the 2-D problems considered in this paper the maximum value of k is 8), (4) t_{fp}: time for a floating-point operation (at the source code level), (5) t_s: setup time for message passing, and (6) t_w: time to transfer a single word between two PEs.

5. Krylov Algorithms on Parallel Machines

Krylov subspace algorithms are fast and robust for the solution of unstructured and nonsymmetric matrix problems. One of these Krylov techniques, the conjugate gradient squared (CGS) algorithm [14], which can be directly applied to complex matrices, is used in the experiments reported here. The CGS algorithm for the solution of the linear system $\mathbf{Ax} = \mathbf{b}$ is shown in Figure 3. An initial guess for the solution vector $\mathbf{x}_0$ and an arbitrarily chosen vector $\tilde{\mathbf{r}}_0$ such that $\tilde{\mathbf{r}}_0^H \mathbf{r}_0 \neq 0$ are input to the algorithm, in addition to the vector $\mathbf{b}$ and the sparse matrix $\mathbf{A}$. The convergence criterion is based on the error measure, $\mathbf{e}_n$.

All Krylov algorithms require a basic linear algebra subroutine kernel that implements vector-vector operations, vector inner products, and matrix-vector multiplication. The vector inner product and matrix-vector multiplication operations need inter-PE communications on a distributed memory parallel machine. The $\mathbf{A}$ matrix is represented using a parallel version of the *modified sparse row* format [12]. In this format, each row of the matrix is represented as a variable length array. Each element of this array is a record containing an element of $\mathbf{A}$ and the corresponding column index.

For simplicity, assume that the matrix dimension N is a multiple of P. Let the $\mathbf{A}$ matrix and the vectors be distributed over the PEs in row *striped* format, where each PE gets N/P contiguous rows, as shown in Figure 4; in particular, PE i gets rows $(N/P)i$ to $(N/P)(i+1)-1$. With the data distribution given in Figure 4, any element-wise vector-vector operation can be performed concurrently by all PEs

(1) $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$; $\mathbf{q}_0 = \mathbf{p}_{-1} = \mathbf{0}$;
(2) $\rho_{-1} = 1$; $n = 0$; $\rho_0 = \tilde{\mathbf{r}}_0^H \mathbf{r}_0$;
(3) **while** (not converged) **do**
(4) $\beta = \rho_n / \rho_{n-1}$;
(5) $\mathbf{u}_n = \mathbf{r}_n + \beta \mathbf{q}_n$;
(6) $\mathbf{p}_n = \mathbf{u}_n + \beta(\mathbf{q}_n + \beta \mathbf{p}_{n-1})$;
(7) $\mathbf{v}_n = \mathbf{A}\mathbf{p}_n$;
(8) $\sigma = \tilde{\mathbf{r}}_0^H \mathbf{v}_n$; $\alpha = \rho_n / \sigma$;
(9) $\mathbf{q}_{n+1} = \mathbf{u}_n - \alpha \mathbf{v}_n$;
(10) $\mathbf{f}_{n+1} = \mathbf{u}_n + \mathbf{q}_{n+1}$;
(11) $\mathbf{r}_{n+1} = \mathbf{r}_n - \alpha \mathbf{A}\mathbf{f}_{n+1}$;
(12) $\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha \mathbf{f}_{n+1}$;
(13) $n = n + 1$;
(14) $\rho_n = \tilde{\mathbf{r}}_0^H \mathbf{r}_n$; $e_n = \mathbf{r}_n^H \mathbf{r}_n$;
(15) **endwhile**

Figure 3. The conjugate gradient squared (CGS) algorithm.

without inter-PE communication. Let the time for an element-wise vector-vector operation be t_{vop}. For the row striped format,

$$t_{vop} = (\frac{N}{P}) t_{fp}. \tag{7}$$

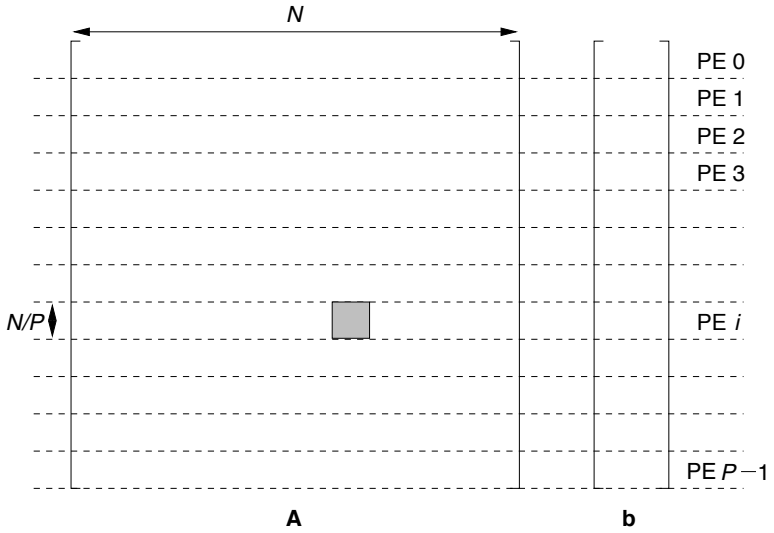

Figure 4. Distribution of the **A** matrix and vectors under the row striped format.

For the inner-products, the evaluation is done in two phases. In the first phase, each PE computes a local inner product on its N/P elements. Because each PE has N/P elements and for each element a multiplication and addition operation are performed, for π local inner products the computation time is $2((\pi N)/P)t_{fp}$. In the next phase, the local inner products are combined to form a global sum. The time taken for this phase is dependent on the interconnection network used by the parallel system. In the Intel Paragon, the PEs are organized in a $\sqrt{P} \times \sqrt{P}$ mesh. Figure 5, shows one way to perform the combining sequence for a sum-to-one-PE operation in a 4×4 mesh (without the wormhole routing). The labels on the arrows are step numbers (e.g., 0 indicates the transfers performed in the initial step). Once the sum is accumulated in a single PE, it is distributed to all the PEs. This distribution is performed by reversing the sequence of operations used to obtain the sum at one PE. Hence four more steps are required to distribute the sum back to all PEs.

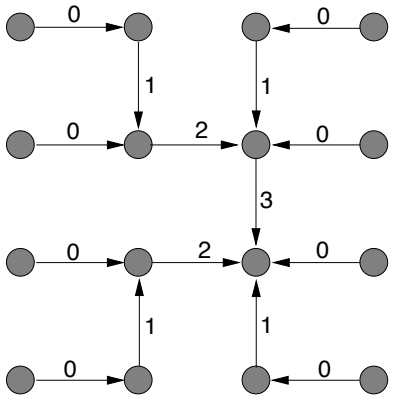

Figure 5. An example global summing sequence for a 4×4 mesh.

In general, for a $\sqrt{P} \times \sqrt{P}$ mesh, the global summing and distribution can be performed in $2\sqrt{P}$ steps, where each step takes $(t_s + \pi t_w)$. Therefore, on the Paragon the combining phase to form the global sum takes $2(t_s + \pi t_w)\sqrt{P}$. Let the time for π vector inner-products be $t_{inner}(\pi)$, which becomes

$$t_{inner}(\pi) = 2(\frac{\pi N}{P})t_{fp} + 2(t_s + \pi t_w)\sqrt{P}. \tag{8}$$

The time taken for a matrix-vector multiplication operation depends on the sparsity pattern of the $\mathbf{A}$ matrix. Consider the situation when $\mathbf{A}$ has an unstructured sparsity pattern and is distributed among the PEs using the row striped format of Figure 4. The shaded square in Figure 4 represents those elements of $\mathbf{A}$ in rows $(N/P)i$ to $(N/P)(i+1)-1$ and columns $(N/P)i$ to $(N/P)(i+1)-1$. To perform the matrix-vector multiplication operation $\mathbf{Ax}$, PE i needs the values of row r of $\mathbf{x}$ if column r of any of rows $(N/P)i$ to $(N/P)(i+1)-1$ of $\mathbf{A}$ contains a non-zero element. Thus, given PE i contains rows $(N/P)i$ to $(N/P)(i+1)-1$ of $\mathbf{x}$, PE

i needs to perform an inter-PE communication for each column j of $\mathbf{A}$ that has at least one non-zero element and $(N/P)(i+1) \leq j$ or $j < (N/P)i$ (i.e., for all non-zero elements of $\mathbf{A}$ that are in PE i and lie outside the shaded area in Figure 4). Because $\mathbf{A}$ is unstructured, PE i could potentially need elements of $\mathbf{x}$ from all other PEs to perform the matrix-vector multiplication.

For the complexity analysis, the worst-case situation of PE i needing $\mathbf{x}$ vector elements from all other PEs is considered. Therefore, an all-to-all broadcast of the $\mathbf{x}$ vector elements is required before each matrix-vector multiplication operation. In a 2-D mesh network of PEs, the all-to-all broadcast can be performed in two phases. In the first phase, each PE broadcasts its N/P elements of the $\mathbf{x}$ vector along the columns of PEs of the 2-D mesh. This operation requires $\sqrt{P}-1$ steps and each step takes $t_s + (N/P)t_w$ time. At the end of this phase, each PE will hold $(N/P)\sqrt{P} = N/\sqrt{P}$ elements of the $\mathbf{x}$ vector. The next phase is similar, but the broadcast is performed along the rows of PEs of the 2-D mesh and each PE sends the $N/\sqrt{P}$ elements it currently holds. The time taken for the second phase is $(t_s + (N/\sqrt{P})t_w)(\sqrt{P}-1)$. Therefore, the total communication time is $t_{mcomm} = (t_s + (N/P)t_s)(\sqrt{P}-1) + (t_s + (N/\sqrt{P})t_s)(\sqrt{P}-1)$. The computation time per row of the $\mathbf{A}$ matrix is k multiplications and $k-1$ additions, thus, for the whole matrix the computation time $t_{mcomp} = (2k-1)(N/P)t_{fp}$. From the above results, the time for matrix-vector multiplication for an unstructured $\mathbf{A}$ matrix is t_{umult} ($u$nstructured $mult$iplication):

$$\begin{aligned} t_{umult} &= t_{mcomp} + t_{mcomm} \\ &= (2k-1)(\frac{N}{P})t_{fp} + 2t_s(\sqrt{P}-1) + \frac{N(P-\sqrt{P})t_w}{P}. \end{aligned} \tag{9}$$

For large P,

$$t_{umult} = (2k-1)(\frac{N}{P})t_{fp} + 2t_s\sqrt{P} + Nt_w. \tag{10}$$

The analysis of the algorithms provided in this paper assumes a large value for P, the number of PEs.

The unstructured $\mathbf{A}$ matrix can be structured to obtain a reordered $\mathbf{A}$ matrix with a banded sparsity pattern. Let the reordered $\mathbf{A}$ matrix have an odd bandwidth of β (i.e., $a_{ij} = 0$ for $|i-j| > \lfloor \beta/2 \rfloor$), then PE i needs to communicate with PEs that are numbered in the range $[i - \lceil \lfloor \beta/2 \rfloor / N/P \rceil ... i + \lceil \lfloor \beta/2 \rfloor / N/P \rceil]$ to obtain the necessary $\mathbf{x}$ vector elements to perform the matrix-vector multiplication. Instead of performing a broadcast operation to retrieve the $\mathbf{x}$ vector elements before every matrix-vector multiplication, for the banded $\mathbf{A}$ matrices, `send` and `receive` type communications are used. Hence, the banded structure of the reordered $\mathbf{A}$ matrix reduces the communication time spent in retrieving the necessary $\mathbf{x}$ vector elements, but the time spent on computation remains the same as the unstructured case.

For very large and very sparse reordered matrices such as those encountered in the application that is considered here, β is such that $(\beta P)/N \leq c$ for a small constant c. For the reordered matrices and the number of PEs considered here, c is equal to two (i.e., PE i needs to perform inter-PE transfers with PEs $i-1$ and

$i+1$). If β does not satisfy this condition, then reordered $\mathbf{A}$ is considered to have an unstructured sparsity pattern and broadcast operations are used instead of the `send` and `receive` calls to retrieve the `x` vector elements.

The total multiplication time for this case is called t_{bmult} (*b*anded *mult*iplication), which is given by

$$t_{bmult} = (2k-1)(\frac{N}{P})t_{fp} + 2t_s + \beta t_w. \tag{11}$$

The parameters derived above are used to obtain expressions for the parallel run time per iteration.

The amount of work done per iteration of the algorithm is approximated by the body of the "while" loop (in Figure 3 for example), i.e., the initialization overhead prior to loop entry is considered negligible. In the CGS algorithm of Figure 3, it can be observed that for each iteration of the "while" loop three inner products and two matrix-vector multiplications are required. Two of the inner products can be performed with a single global summing operation, i.e., the local inner products corresponding to the two inner products in Line (14) of Figure 3 can be reduced in a single global operation (with two operands). Vector-vector additions, vector-vector subtractions, and vector-scalar multiplications are also needed to implement the algorithm. However, these operations are completely parallelizable, i.e., no inter-PE communication is involved in a distributed memory machine.

The following expressions for the parallel run time of the CGS algorithm are derived assuming a data distribution as described in the beginning of this section. Table 1 shows the line by line time complexity of the CGS algorithm. Note that only the expressions within the `while` loop are considered here.

Table 1. Line by line time complexity of the CGS algorithm

line number	*time complexity*
(4)	t_{fp}
(5)	$2(N/P)t_{fp}$
(6)	$4(N/P)t_{fp}$
(7)	t_{mult}
(8)	$t_{fp} + t_{inner}(1)$
(9)	$2(N/P)t_{fp}$
(10)	$(N/P)t_{fp}$
(11)	$t_{mult} + 2(N/P)t_{fp}$
(12)	$2(N/P)t_{fp}$
(13)	t_{fp}
(14)	$t_{inner}(2)$

Let the time taken for the ith iteration of the CGS algorithm be t^i_{CGS}. An expression for the value of t^i_{CGS} can be obtained by counting the different types of operations performed per iteration of the CGS algorithm. Therefore,

$$t^i_{CGS} = (3 + \frac{13N}{P})t_{fp} + t_{inner}(1) + t_{inner}(2) + 2t_{mult}, \tag{12}$$

where the $(3 + 13N/P)t_{fp}$ term comes from Lines (4), (5), (6), (8), (9), (10), (11), (12), and (13), the $t_{inner}(1)$ term comes from Line (8), the $t_{inner}(2)$ term comes from Line (14), and the $2t_{mult}$ term comes from Lines (7) and (11) of Figure 3. The number of PEs that result in the minimum parallel execution time, P_{CGS}, can be derived by solving for P in $(\partial t^i_{CGS})/(\partial P) = 0$.

If $\mathbf{A}$ has an unstructured sparsity pattern, then Equation (12) can be simplified by substituting the value of t_{inner} from Equation (8) and the value of t_{umult} for t_{mult} from Equation (10). For an unstructured $\mathbf{A}$ matrix, let the time per iteration of the CGS algorithm be $t^i_{CGS\_u}$.

$$t^i_{CGS\_u} = (8t_s + 6t_w)\sqrt{P} + \frac{(17 + 4k)Nt_{fp}}{P} + 3t_{fp} + 2Nt_w \tag{13}$$

The number of PEs that result in the minimum parallel execution time, P_{CGS_u}, can be derived as the following by solving for P in $(\partial t^i_{CGS\_u})/(\partial P) = 0$.

$$P_{CGS\_u} = \left(\frac{(17 + 4k)Nt_{fp}}{4t_s + 3t_w}\right)^{2/3} \tag{14}$$

Similarly, for a reordered $\mathbf{A}$ (obtained by reordering the unstructured $\mathbf{A}$) with a bandwidth of β, let the time per iteration of the CGS algorithm be $t^i_{CGS\_b}$ and the number of PEs for minimum parallel execution time be P_{CGS_b}. A general expression for the time taken per iteration of the CGS algorithm is given by Equation (12), i.e., the expression is applicable for reordered or unstructured $\mathbf{A}$ matrices. The only parameter in Equation (12) that depends on the structure of the $\mathbf{A}$ matrix is the t_{mult} term. By substituting the value of t_{bmult} given by Equation (11) into Equation (12) an expression for $t^i_{CGS\_b}$ can be obtained. Therefore, from Equations (8), (11), and (12):

$$t^i_{CGS\_b} = (4t_s + 6t_w)\sqrt{P} + \frac{(17 + 4k)Nt_{fp}}{P} + 3t_{fp} + 4t_s + 2\beta t_w \tag{15}$$

The number of PEs that result in the minimum parallel execution time, P_{CGS_b}, can be derived as the following by solving for P in $(\partial t^i_{CGS\_b})/(\partial P) = 0$.

$$P_{CGS\_b} = \left(\frac{(17 + 4k)Nt_{fp}}{2t_s + 3t_w}\right)^{2/3} \tag{16}$$

6. The Modified CGS (MCGS) Algorithm

In Figure 3, The element-wise vector-vector, vector-scalar, scalar-scalar, and matrix-vector operations that follow the inner products in Lines (8) and (14) depend on the values generated by the inner product operations. Therefore, the operation in Lines (8) and (14) form two distinct synchronization operations per iteration of the "while" loop. Depending on the values of N and P, the synchronization overhead can cause a significant impact on the actual execution time. The synchronization overhead can be reduced by merging the inner products together so that a single global summing is sufficient for an iteration of the "while" loop.

The basic idea in formulating the MCGS algorithm is to merge the inner products present in the CGS algorithm so that they can be evaluated using a single global summing operation per iteration of the "while" loop. One way of merging the inner products is to reformulate the algorithm so that the inner product in Line (8) of Figure 3 can be combined with that in Line (14) of Figure 3. Let $\mathbf{s}_i = \mathbf{A}\mathbf{r}_i$, $\mathbf{w}_i = \mathbf{A}\mathbf{q}_i$, and $\mathbf{y}_i = \mathbf{A}\mathbf{p}_i$. From Figure 3, consider Lines (5), (6), (7), and (8)

$$\begin{aligned}
\sigma &= \tilde{\mathbf{r}}_0^H \mathbf{v}_n \\
&= \tilde{\mathbf{r}}_0^H \mathbf{A}\mathbf{p}_n \\
&= \tilde{\mathbf{r}}_0^H \mathbf{A}(\mathbf{r}_n + 2\beta\mathbf{q}_n + \beta^2\mathbf{p}_{n-1}) \\
&= \tilde{\mathbf{r}}_0^H \mathbf{s}_n + 2\beta\tilde{\mathbf{r}}_0^H \mathbf{w}_n + \beta^2\tilde{\mathbf{r}}_0^H \mathbf{y}_{n-1}
\end{aligned} \tag{17}$$

Consider Line (6) of Figure 3,

$$\begin{aligned}
\mathbf{p}_n &= \mathbf{u}_n + \beta(\mathbf{q}_n + \beta\mathbf{p}_{n-1}) \\
\mathbf{A}\mathbf{p}_n &= \mathbf{A}\mathbf{r}_n + 2\beta\mathbf{A}\mathbf{q}_n + \beta^2\mathbf{A}\mathbf{p}_{n-1} \\
\mathbf{y}_n &= \mathbf{s}_n + 2\beta\mathbf{w}_n + \beta^2\mathbf{y}_{n-1}
\end{aligned} \tag{18}$$

From Line (9) of Figure 3,

$$\begin{aligned}
\mathbf{q}_{n+1} &= \mathbf{u}_n - \alpha\mathbf{v}_n \\
&= \mathbf{u}_n - \alpha\mathbf{A}\mathbf{p}_n \\
&= \mathbf{u}_n - \alpha\mathbf{y}_n
\end{aligned} \tag{19}$$

From Lines (5), (10), and (11) of Figure 3,

$$\begin{aligned}
\mathbf{r}_{n+1} &= \mathbf{r}_n - \alpha\mathbf{A}(\mathbf{u}_n + \mathbf{q}_{n+1}) \\
&= \mathbf{r}_n - \alpha(\mathbf{A}\mathbf{r}_n + \beta\mathbf{A}\mathbf{q}_n + \mathbf{A}\mathbf{q}_{n+1}) \\
&= \mathbf{r}_n - \alpha(\mathbf{s}_n + \beta\mathbf{w}_n + \mathbf{w}_{n+1})
\end{aligned} \tag{20}$$

Using the derivations given in the Equations (17), (18), (19), and (20), the CGS algorithm of Figure 3 can be reformulated as the MCGS algorithm of Figure 6.

In the MCGS algorithm of Figure 6, the inner products are all grouped together so that a single global summing operation (with five operands) is sufficient to evaluate them. The number of matrix-vector multiplications in the MCGS algorithm remain the same as the number of matrix-vector multiplications in the CGS algorithm.

Due to the modifications, additional scalar-scalar, scalar-vector, and element-wise vector-vector operations are performed in the MCGS algorithm compared to the operations performed by the CGS algorithm. However, these additional operations scale linearly with the inverse of the number of PEs.

(1) $\gamma_{-1} = 1$; $\mathbf{p}_0 = \mathbf{0}$; $\mathbf{q}_0 = \mathbf{0}$; $\mathbf{w}_0 = \mathbf{0}$;
(2) $\mathbf{w}_{-1} = \mathbf{0}$; $\mathbf{y}_{-1} = \mathbf{0}$; $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$; $\mathbf{s}_0 = \mathbf{A}\mathbf{r}_0$;
(3) $\gamma_0 = \tilde{\mathbf{r}}_0^H \mathbf{r}_0$; $\eta = \tilde{\mathbf{r}}_0^H \mathbf{s}_0$;
(4) $\mu = \tilde{\mathbf{r}}_0^H \mathbf{w}_0$; $\lambda = \tilde{\mathbf{r}}_0^H \mathbf{y}_{-1}$; $\rho_{-1} = 1$; $n = 0$;
(5) **while** (not converged) **do**
(6) $\beta = \gamma_n / \gamma_{n-1}$; $\sigma = \eta + \beta(2\mu + \lambda\beta)$;
(7) $\alpha = \gamma_n / \sigma$;
(8) $\mathbf{u}_n = \mathbf{r}_n + \beta \mathbf{q}_n$;
(9) $\mathbf{y}_n = \mathbf{s}_n + 2\beta \mathbf{w}_n + \beta^2 \mathbf{y}_{n-1}$;
(10) $\mathbf{q}_{n+1} = \mathbf{u}_n - \alpha \mathbf{y}_n$;
(11) $\mathbf{w}_{n+1} = \mathbf{A}\mathbf{q}_{n+1}$;
(12) $\mathbf{r}_{n+1} = \mathbf{r}_n - \alpha(\mathbf{s}_n + \beta \mathbf{w}_n + \mathbf{w}_{n+1})$;
(13) $\mathbf{s}_{n+1} = \mathbf{A}\mathbf{r}_{n+1}$;
(14) $\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha(\mathbf{u}_n + \mathbf{q}_{n+1})$;
(15) $n = n + 1$;
(16) $\gamma_n = \tilde{\mathbf{r}}_0^H \mathbf{r}_n$; $\eta = \tilde{\mathbf{r}}_0^H \mathbf{s}_n$; $\mu = \tilde{\mathbf{r}}_0^H \mathbf{w}_n$; $\lambda = \tilde{\mathbf{r}}_0^H \mathbf{y}_{n-1}$; $e_n = \mathbf{r}_n^H \mathbf{r}_n$;
(17) **endwhile**

Figure 6. The modified conjugate gradient squared (MCGS) algorithm

The following expressions for the parallel run time of the MCGS algorithm are derived assuming a data distribution as described in the beginning of Section 5. Table 2 shows the line by line time complexity of the MCGS algorithm. Note that only the expressions within the `while` loop are considered here.

The parallel run time for the i-th iteration of the MCGS algorithm is

$$t^i_{MCGS} = (10 + \frac{16N}{P})t_{fp} + t_{inner}(5) + 2t_{mult}, \tag{21}$$

where the $(10 + 16N/P)t_{fp}$ term comes from Lines (6), (7), (8), (9), (10), (12), (14), and (15), the $t_{inner}(5)$ term comes from Lines (16), and the $2t_{mult}$ term comes from Lines (11) and (13) of Figure 6. The number of PEs that result in the minimum parallel execution time, P_{MCGS}, can be derived by solving for P in $(\partial t^i_{MCGS})/(\partial P) = 0$.

If $\mathbf{A}$ has an unstructured sparsity pattern, then Equation (21) can be simplified by substituting the value of t_{inner} from Equation (8) and the value of t_{umult} for t_{mult} from Equation (10), giving

$$t^i_{MCGS\_u} \;=\; (6t_s + 10t_w)\sqrt{P} + \frac{(24 + 4k)Nt_{fp}}{P} + 10t_{fp} + 2Nt_w \tag{22}$$

Table 2. Line by line time complexity of the MCGS algorithm

line number	*time complexity*
(6)	$6t_{fp}$
(7)	t_{fp}
(8)	$2(N/P)t_{fp}$
(9)	$2t_{fp} + 4(N/P)t_{fp}$
(10)	$2(N/P)t_{fp}$
(11)	t_{mult}
(12)	$5(N/P)t_{fp}$
(13)	t_{mult}
(14)	$3(N/P)t_{fp}$
(15)	t_{fp}
(16)	$t_{inner}(5)$

$$P_{MCGS\_u} = \left(\frac{(24+4k)Nt_{fp}}{3t_s + 5t_w}\right)^{2/3} \tag{23}$$

Similarly, for a banded $\mathbf{A}$ (obtained by reordering the unstructured $\mathbf{A}$) with a bandwidth of β, from Equations (8) and (11),

$$t^i_{MCGS\_b} = (2t_s + 10t_w)\sqrt{P} + \frac{(24+4k)Nt_{fp}}{P} + 10t_{fp} + 4t_s + 2\beta t_w \tag{24}$$

$$P_{MCGS\_b} = \left(\frac{(24+4k)Nt_{fp}}{t_s + 5t_w}\right)^{2/3} \tag{25}$$

7. Comparison of the Algorithms

From the derivation of the MCGS algorithm in Section 6, it can be observed that MCGS and CGS are basically the same algorithms, but with a different computing sequence. Because both algorithms are doing exactly the same calculations, they are numerically equivalent, unless rounding-off errors create a difference. The experimental results presented in Section 8 further support this claim.

Reordering the matrix $\mathbf{A}$ for bandwidth reduction reduces the time taken for the matrix-vector multiplication operation, but the time for the other operations remains almost the same. Therefore, reordering the $\mathbf{A}$ matrix reduces the total execution time. Because the total execution time for the banded $\mathbf{A}$ matrix is lower than the execution time for the unstructured $\mathbf{A}$ matrix case, the inner product operation contributes a bigger percentage towards the overall execution time. Hence,

the reduction in the global synchronization overhead is likely to have a more significant impact for banded **A** matrices.

Because MCGS and CGS take the same number of iterations to converge, the parallel run time of the algorithms can be compared using the time taken per iteration of the algorithm, i.e., for MCGS to perform better than CGS, t^i_{MCGS} should be less than t^i_{CGS}. Consider the case where **A** is banded with a bandwidth β and $t_s \gg t_w$, so the t_w term can be ignored. Using the expressions derived in Equations (15) and (24) for $t^i_{CGS\_b}$ and $t^i_{MCGS\_b}$, respectively, for MCGS to perform better than CGS

$$P > \left(\frac{7Nt_{fp}}{2t_s}\right)^{2/3} \text{ for } t_s \gg t_w. \tag{26}$$

Let $t^{min}_{MCGS\_b}$ and $t^{min}_{CGS\_b}$ be the minimum parallel execution times for the MCGS and CGS algorithms, respectively. The value of P_{CGS_b} can be substituted from Equation (16) into Equation (15) to obtain the following expression for $t^{min}_{CGS\_b}$:

$$\begin{aligned} t^{min}_{CGS\_b} &= (4t_s + 6t_w)\left[\sqrt{P_{CGS\_b}} + \frac{1}{2}\left(\frac{(17+4k)Nt_{fp}}{2t_s+3t_w}\right)\frac{1}{P_{CGS\_b}}\right] \\ &\quad +3t_{fp} + 4t_s + 2\beta t_w \\ &= (6t_s + 9t_w)\sqrt{P_{CGS\_b}} + 3t_{fp} + 4t_s + 2\beta t_w \end{aligned} \tag{27}$$

Similarly, Equations (24) and (25) can be used to derive the expression for $t^{min}_{MCGS\_b}$:

$$\begin{aligned} t^{min}_{MCGS\_b} &= (2t_s + 10t_w)\left[\sqrt{P_{MCGS\_b}} + \frac{1}{2}\left(\frac{(24+4k)Nt_{fp}}{t_s+5t_w}\right)\frac{1}{P_{MCGS\_b}}\right] \\ &\quad +10t_{fp} + 4t_s + 2\beta t_w \\ &= (3t_s + 15t_w)\sqrt{P_{MCGS\_b}} + 10t_{fp} + 4t_s + 2\beta t_w \end{aligned} \tag{28}$$

For distributed memory MIMD machines, where $t_s \gg t_w$, from Equations (27) and (28), $t^{min}_{CGS\_b}$ and $t^{min}_{MCGS\_b}$ can be approximated as follows:

$$t^{min}_{CGS\_b} = 6t_s\sqrt{P_{CGS\_b}} \tag{29}$$

$$t^{min}_{MCGS\_b} = 3t_s\sqrt{P_{MCGS\_b}} \tag{30}$$

$$\begin{aligned} \frac{t^{min}_{MCGS\_b}}{t^{min}_{CGS\_b}} &= \frac{3}{6}\sqrt{\frac{P_{MCGS\_b}}{P_{CGS\_b}}} \\ &= \frac{1}{2}\left[\frac{(24+4k)Nt_{fp}(2t_s+3t_w)}{(t_s+5t_w)(17+4k)Nt_{fp}}\right]^{1/3} \\ &\approx \left(\frac{(24+4k)}{4(17+4k)}\right)^{1/3} = 0.66 \text{ for } k = 8 \end{aligned} \tag{31}$$

Hence, the best MCGS timing is approximately up to 34% better than the best CGS timing. The variation of the ratio $t^{min}_{MCGS\_b}/t^{min}_{CGS\_b}$ with the value of k is shown in Figure 7. Also, for the assumptions stated, Equation (26) provides the machine size (in number of PEs) above which MCGS performs better than CGS. This can be used to automatically select an algorithm (either CGS or MCGS) depending on N, P, t_{fp}, and t_s [13, 18].

This approach will allow heterogeneous computing management systems [9] to adaptively select the best algorithm to use from a set of algorithms. In this case, the problem size (N) is fixed, but the values of P, t_s, and t_{fp} can vary depending on the machine that is selected for executing the solver. For the heterogeneous computing management systems to make the best decision, it is necessary to provide information such as that in Equation (26) to the management systems.

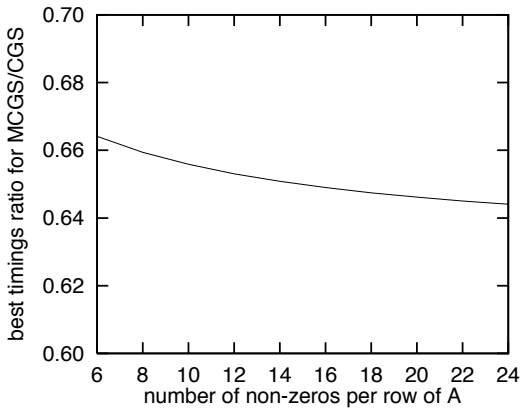

Figure 7. Variation of the $t^{min}_{MCGS\_b}/t^{min}_{CGS\_b}$ ratio with the value of k

8. Numerical Experiments and Discussion

Experiments were conducted on up to 128-nodes of the Purdue mesh-connected Intel Paragon XP/S [2] and up to 16 thin nodes of the Purdue multistage network connected IBM SP2 [1, 2, 16]. The algorithms were implemented on the Paragon using C and NX message passing library routines. The algorithms were implemented on the IBM SP2 using C and message passing interface (MPI) library routines. The global summing operations are performed using the recursive doubling routines provided by the NX and MPI libraries.

The timings varied slightly (less than five percent) between trials. This is due to cache misses, background network traffic, and possibly other factors. The minimum of the timing values was taken as the representative value, because the minimum occurs when the impact due to external factors is minimal.

Two electromagnetic scattering matrices were used in the experiments reported in this section. The bigger matrix that is referred to as $\mathbf{A}_1$ is 36818-by-36818 and the

smaller one that is referred to as $\mathbf{A}_2$ is 2075-by-2075. $\mathbf{A}_1$ is for a 5λ radius circular computation domain, a rectangular $6 \times 0.1\lambda$ perfect conductor scatterer, and TM polarization with an incidence angle of 45° with respect to the plate normal. $\mathbf{A}_2$ is for an incident angle of 270°. Each matrix was reordered to form a banded matrix by a reverse Cuthill-McKee algorithm [6], obtaining two more matrices $\mathbf{A}_{1b}$ and $\mathbf{A}_{2b}$, where $\mathbf{A}_{1b}$ is from $\mathbf{A}_1$ and $\mathbf{A}_{2b}$ is from $\mathbf{A}_2$. The sparsity pattern of $\mathbf{A}_1$ is shown in Figure 8 and $\mathbf{A}_{1b}$ is shown in Figure 9. The thick straight line shown in Figure 9 denotes a band that contains all the non-zero elements of the reordered $\mathbf{A}$ matrix. The timings for the reordering phase are not reported here, because these timings do not affect the CGS versus MCGS comparison.

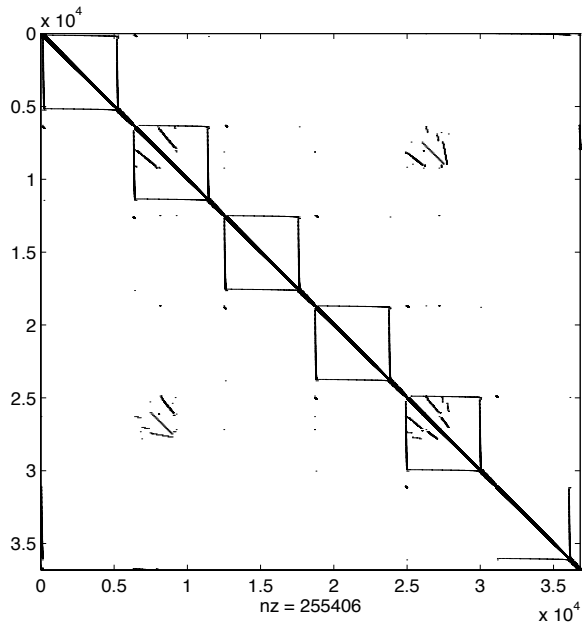

Figure 8. Sparsity pattern of test matrix $\mathbf{A}_1$, $N = 36818$.

The execution time of the CGS algorithm is dependent on the time taken for the basic operations: matrix-vector multiplication (referred as `matvec`), inner-products (referred as `innerprod`), element-wise vector-vector operations, scalar-vector operations, and scalar-scalar operations. The last three do not involve inter-PE communication (i.e., each PE executes them independently) and will be collectively referred to as `indops`. Thus, there are three categories to be considered: `matvec`, `innerprod`, and `indops`. To get a better understanding of the variation in the performance with the number of PEs, the timings were measured separately for `innerprod`, `matvec`, and `indops`. Figure 10 shows the three timing curves corresponding to `innerprod` time, `matvec` time, and `indops` time for the Paragon.

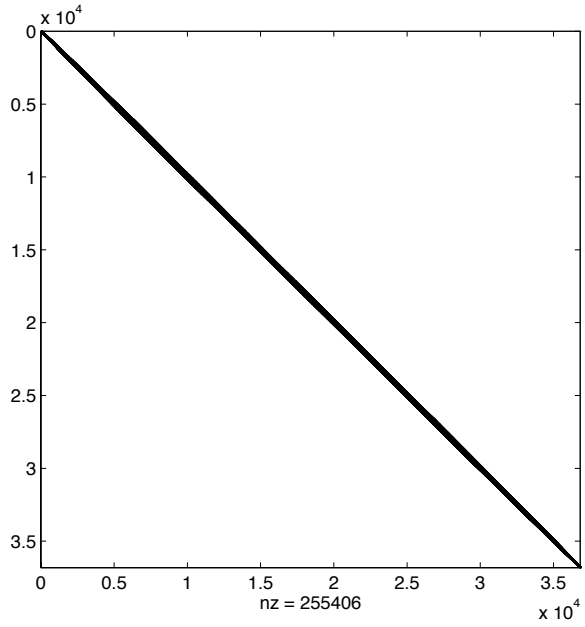

Figure 9. Sparsity pattern of test matrix $\mathbf{A}_{1b}$ obtained by reordering $\mathbf{A}_1$ using the reverse Cuthill-McKee algorithm.

The `innerprod` time includes the time needed to compute the local product terms and the time needed to sum the local products. The computation time for the local products decreases with increasing P, whereas the global summing time increases with increasing P. Likewise, the `matvec` time includes the time to do the multiplications and additions and the inter-PE communication time needed to retrieve remote vector values. From Figure 10 it can be noted that the timing for `matvec` decreases with increasing P. This is due to the decrease in the computation time associated with the `matvec` operation. Similarly, the time for the `indops` decreases with increasing number of PEs.

For the larger matrix ($\mathbf{A}_1$) and $4 \leq P \leq 16$, the `indops` time is the dominant component of the total execution time. Therefore, any optimizations done for the `innerprod` and `matvec` times will not have a significant impact on the overall execution time. However, as the number of PEs are increased the `indops` time decreases and at 128 PEs the `innerprod` and `matvec` times are nearly as same as the `indops` time. Reordering the matrix $\mathbf{A}_1$ has a reducing effect on the `matvec` time, as shown in Figure 11 for the Paragon. The `indops` and `innerprod` times are independent of the reordering. The overall execution time of the CGS algorithm is reduced, due to the reordering of $\mathbf{A}$ for bandwidth reduction. For the bandwidth-reduced $\mathbf{A}$, `innerprod` time forms a higher percentage of the overall execution time compared to the situation where matrix $\mathbf{A}$ is unstructured. Therefore, the optimizations towards reducing the `innerprod` are likely to have a relatively high

impact on the overall execution time when the matrix $\mathbf{A}$ is reordered for bandwidth reduction.

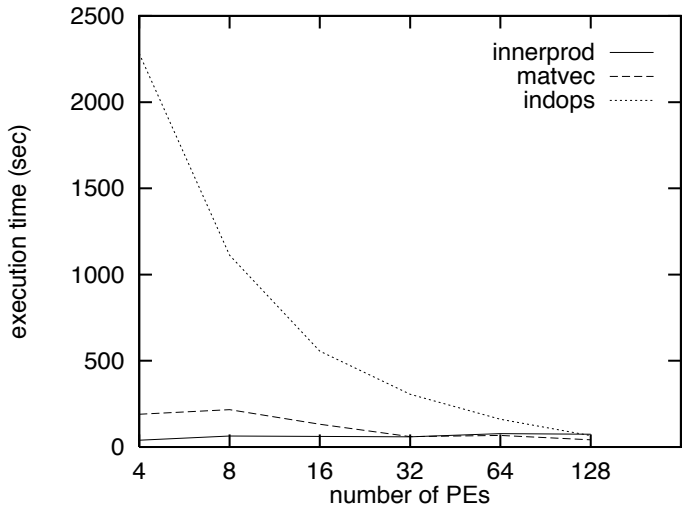

Figure 10. Independent components of the total execution time for CGS on the Intel Paragon for $\mathbf{A}_1$.

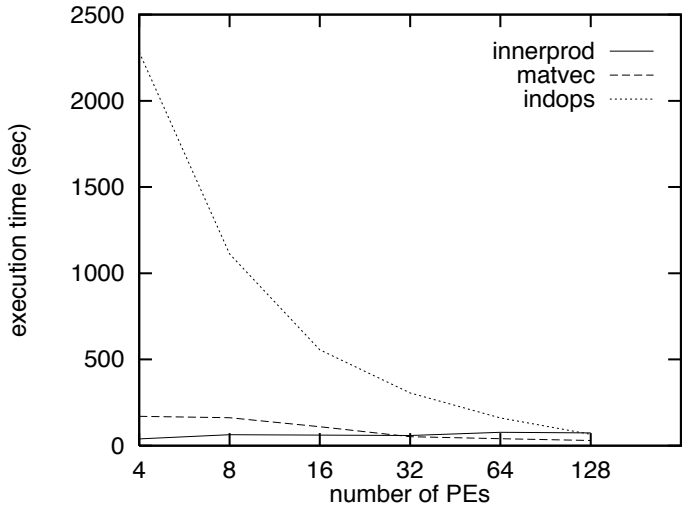

Figure 11. Independent components of the total execution time for CGS on the Intel Paragon for $\mathbf{A}_{1b}$.

From the discussion of MCGS in Section 6, it is evident that the modifications performed in merging one inner product with the other one in the CGS algorithm

to formulate the MCGS algorithm has introduced additional element-wise vector operations, vector-scalar, scalar-scalar operations (i.e., `indops` forms a higher percentage of the total number of operations performed by the MCGS algorithm compared to the `indops` performed by the CGS algorithm). For smaller P and larger N values, the `indops` time dominates the other two components, `innerprod` and `matvec`. Therefore, for smaller P and larger N values, MCGS takes more time than the CGS algorithm, as shown in Figure 12 with $\mathbf{A}_{1b}$ for the Paragon. As P is increased MCGS starts to perform better. The same comparison is shown for $\mathbf{A}_{2b}$ in Figure 13. The experimentally derived difference of 20% confirms a significant improvement using MCGS for this set of parameters.

Equation (26) is used to predict the value of P^*, where P^* is the number of PEs beyond which the MCGS algorithm should outperform the CGS algorithm. The values of N, P, t_s, and t_{fp} are needed to predict the value of P^*. To determine the value of t_{fp}, small code segments with floating-point operations were executed on the Paragon and the executions were timed. From the experiments, the value of t_{fp} was found to be equal to 1.078×10^{-6} seconds. The value of t_{fp} determined by these experiments do not include any loop or other integer operation overhead. However, the time delays incurred by cache misses are included in the measured value of t_{fp}.

The value of t_s is measured by an echo server experiment, as shown in Figure 14. In the echo server experiment, two PEs which are adjacent to each other are selected. One PE runs the echo server, which is a program that retransmits whatever it receives back to the sending PE. The other PE in the experiment executes the testing program. The testing program turns on a timer and sends a zero-length message to the PE that runs the echo server. Once the testing PE receives the reply back from the echo server, the timer is stopped and the elapsed time is measured. The elapsed time is equal to two times the message setup overhead, i.e., $2t_s$. The value of t_s was determined as $112\,\mu s$ from the echo server experiment for the Intel Paragon. For the Paragon, using the values of t_s and t_{fp} determined by the above experiments, the value of P^* is estimated by Equation (26) to be 115 for $\mathbf{A}_{1b}$ and 17 for $\mathbf{A}_{2b}$.

In Figure 15, the MCGS algorithm is compared with the CGS algorithm on the IBM SP2 for $\mathbf{A}_{2b}$. The PEs in the SP2 are faster and have larger cache memory than the PEs in the Paragon, and the SP2 has a relatively smaller t_{fp}/t_s ratio than the Paragon. Therefore, in the SP2, the MCGS algorithm is the better algorithm for even smaller number of PEs, as compared to the Intel Paragon. A complexity analysis of the algorithms was not performed for the SP2 machine, because only empirically derived communication models are available for the SP2 [19, 20]. From Figure 16, it can be observed that for matrix $\mathbf{A}_{1b}$ that CGS outperforms MCGS, however, as the number of PEs are increased the performance gap between the MCGS and CGS decreases. At a sufficiently large P, the MCGS algorithm should perform better than the CGS algorithm for larger matrices (e.g., $\mathbf{A}_{1b}$) as it did for smaller matrices (e.g., $\mathbf{A}_{2b}$).

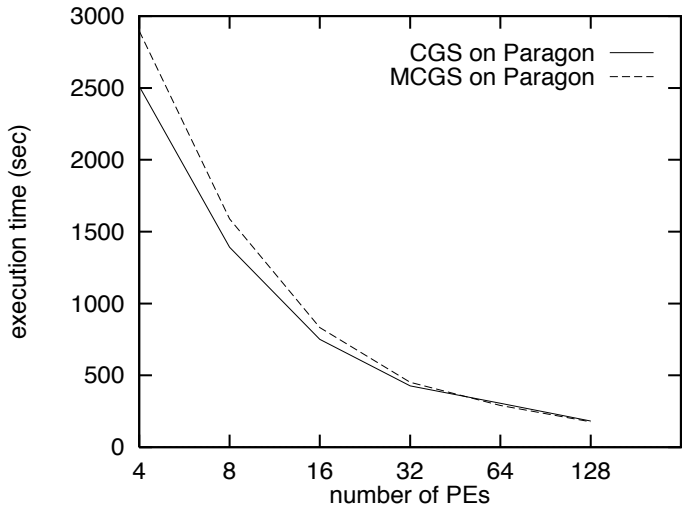

Figure 12. Execution time of CGS versus MCGS for $\mathbf{A}_{1b}$ on the Paragon.

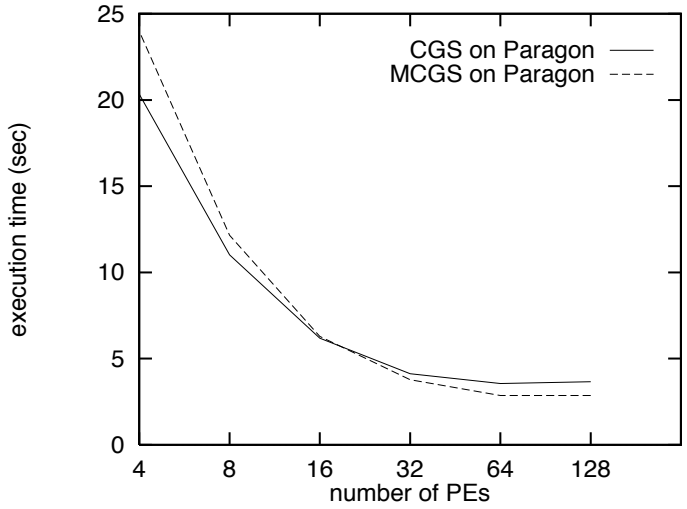

Figure 13. Execution time of CGS versus MCGS for $\mathbf{A}_{2b}$ on the Paragon.

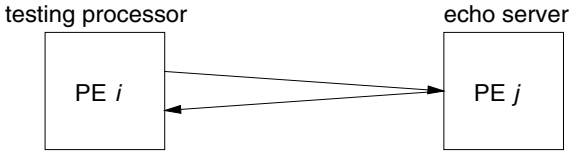

Figure 14. The PE configuration for the echo server experiment.

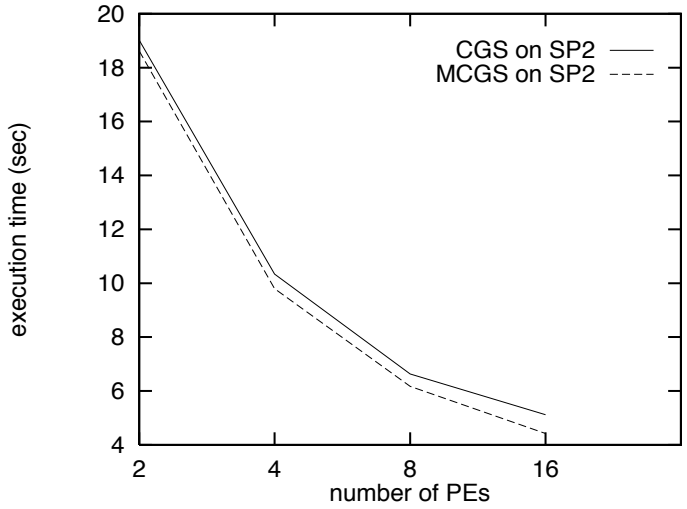

Figure 15. Execution time of CGS versus MCGS for $\mathbf{A}_{2b}$ on the SP2.

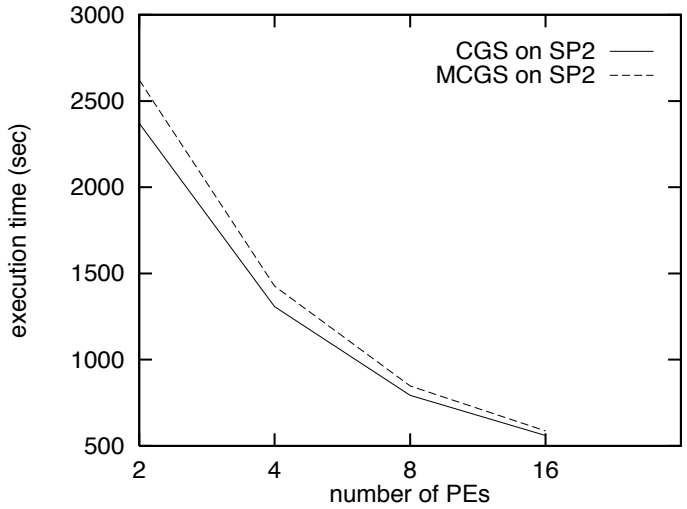

Figure 16. Execution time of CGS versus MCGS for $\mathbf{A}_{1b}$ on the SP2.

9. Conclusions

A reformulation of the CGS algorithm called the MCGS is presented in this paper. The system of linear equations obtained from finite element modeling of open-region electromagnetic problems was solved using the CGS and MCGS algorithms. This work examined ways of reducing the communication and synchronization overhead associated with implementing the CGS algorithm on MIMD parallel machines.

An experimental and theoretical complexity analysis was performed to evaluate the performance benefits. The approximate complexity analysis of the CGS and MCGS algorithms on a mesh-based multiprocessor model estimates that the performance of the MCGS algorithm may be up to 34% better than the performance of the CGS algorithm, depending on the machine architecture. The experimental results obtained from a 128-processor Intel Paragon show that the performance of the MCGS algorithm is at least 20% better than the performance of the CGS algorithm for a 2075×2075 sparse matrix. The experimental results on the IBM SP2 indicate that the MCGS is the better algorithm for the 2075×2075 sparse matrix and the CGS algorithm is the better algorithm for the 36818×36818 sparse matrix. The MCGS can be expected to be the better algorithm for the larger matrix as the number of PEs increases. Any optimization or preconditioning that is used on CGS to reduce the overall computation time can be used on MCGS as well. The techniques used here to reduce the synchronization overhead of CGS can be extended to other nonsymmetric Krylov methods, such as the BiCGSTAB algorithm [17].

Because MCGS is not the best method for all situations, the use of a set of algorithms approach is proposed. In the set of algorithms approach, either CGS or MCGS is selected depending on the values of N and P. The set approach provides an algorithm that is more scalable than either the CGS or MCGS algorithms alone. Conditions such as the one developed here to choose between CGS or MCGS depending on the system parameters are also useful in the area of HC mapping systems [9]. In HC mapping, the input data (e.g., matrix size) remains fixed, but the system parameters are varied, i.e., the mapping system estimates the performance on different machines and executes the application on the machine that is expected to yield the best performance. To obtain the best mapping, it is necessary for the HC mapping systems to have information such as those provided by the conditions to select either MCGS or CGS depending on system parameters.

Acknowledgments

A preliminary version of this paper was presented at the 1998 International Conference on Parallel Processing. The authors thank the referees for their suggestions. This work was supported in part by the DARPA/ITO Quorum Program under NPS subcontract numbers N62271-97-M-0900 and N62271-98-M-0217.

References

1. T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal,* 34:152-184, 1995.
2. G. S. Almasi and A. Gotlieb. *Highly Parallel Computing,* 2nd ed. Benjamin Cummings, Redwood City, CA, 1994.
3. E. Dazevedo, V. Eijkhout, and C. Romaine. Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors. *LaPack Working Note 56,* 1992.
4. R. W. Freund. Conjugate gradient-type methods for linear systems with complex symmetric coefficient matrices. *SIAM Journal on Scientific and Statistical Computing,* 13:425-448, 1992.
5. R. W. Freund, G. H. Golub, and N. H. Nachtigal. Iterative solution of linear systems. *Acta Numerica,* pp. 57-100, 1992.
6. A. George and J. W. Lu. *Computer Solution of Large Sparse Positive Definite Systems,* Prentice-Hall, NJ, 1981.
7. B. Lichtenberg. Finite element modeling of wavelength-scale diffractive element. PhD thesis, Purdue University, West Lafayette, IN, 1994.
8. B. Lichtenberg, K. J. Webb, D. B. Meade, and A. F. Peterson. Comparison of two-dimensional conformal local radiation boundary conditions. *Electromagnetics,* 16:359-384, 1996.
9. M. Maheswaran, T. D. Braun, and H. J. Siegel. High-performance mixed-machine heterogeneous computing. *6th Euromicro Workshop on Parallel and Distributed Processing,* pp. 3-6, 1998.
10. G. Meurant. Multitasking the conjugate gradient on the Cray X-MP/48. *Parallel Computing,* 5:267-280, 1987.
11. Y. Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing,* 10:1200-1232, 1989.
12. Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. *LaPack Working Note 50,* 1994.
13. H. J. Siegel, L. Wang, J. E. So, and M. Maheswaran. Data Parallel Algorithms. In A. Y. Zomaya, ed., *Parallel and Distributed Computing Handbook,* pp. 466-499. McGraw Hill, New York, NY, 1996.
14. P. Sonnevald. CGS: A fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing,* 10:36-52, 1989.
15. G. Strang. *Linear Algebra and its Applications,* 3rd ed. Harcourt Brace Jovanovich, San Diego, CA, 1988.
16. C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker. The SP2 high-performance switch. *IBM Systems Journal,* 34:185-204, 1995.
17. H. A. Van Der Vorst. Bi-CGSTAB: A fast and smooth converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing,* 12:631-644, 1992.
18. M-C. Wang, W. G. Nation, J. B. Armstrong, H. J. Siegel, S. D. Kim, M. A. Nichols, and M. Gherrity. Multiple quadratic forms: A case study in the design of data-parallel algorithms. *Journal of Parallel and Distributed Computing,* 21:124-139, 1994.
19. Z. Xu and K. Hwang. Modeling communication overhead: MPI and MPL performance on the IBM SP2. *IEEE Parallel and Distributed Technology,* 4:9-23, 1996.
20. Z. Xu and K. Hwang. Early prediction of MPP performance: The SP2, T3D, and Paragon experiences. *Parallel Computing,* 22:917-924, 1996.

Scheduled to appear in The Journal of Supercomputing, , 25 ()

Contributing Authors

Muthucumaru Maheswaran is an Assistant Professor in the Department of Computer Science at the University of Manitoba, Canada. He received a BSc degree from the University of Peradeniya, Sri Lanka and the MSEE and PhD degrees from Purdue University. He received a Fulbright scholarship to pursue his MSEE degree at Purdue University. His research interests include computer architecture, distributed computing, heterogeneous computing, and resource management systems for metacomputing.

Kevin J. Webb received the B.Eng. and M.Eng. degrees from the Royal Melbourne Institute of Technology, Australia, in 1978 and 1983, respectively, the M.S.E.E. degree from the University of California, Santa Barbara, in 1981, and the Ph.D. degree from the University of Illinois, Urbana, in 1984. From 1984 through 1989 he was on the faculty of the University of Maryland, College Park, and since January, 1990 he has been with the School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana. He spent the 1997/98 academic year on sabbatical at the Royal Melbourne Institute of Technology and the Australian National University, Canberra.

Howard Jay Siegel is a Professor in the School of Electrical and Computer Engineering at Purdue. He received two BS degrees from MIT, and the MA, MSE, and PhD degrees from Princeton. He is a Fellow of the IEEE and a Fellow of the ACM. He has coauthored over 250 technical papers, was a Coeditor-in-Chief of the Journal of Parallel and Distributed Computing, and served on the Editorial Boards of both the IEEE Transactions on Parallel and Distributed Systems and the IEEE Transactions on Computers.